Project Number 288094

eCOMPASS
eCO-friendly urban Multi-modal route PlAnning Services for mobile uSers

STREP
Funded by EC, INFSO-G4(ICT for Transport) under FP7

eCOMPASS – TR – 040

Data Trust and Security Mechanisms

D. Kehagias, C. Zaroliagis

June 2013

eCO-friendly urban Multi-modal route PlAnning Services for mobile uSers

FP7 - Information and Communication Technologies

Grant Agreement no: 288094
Collaborative Project
Project start: 1 November 2011, Duration: 36 months

Technical Report: Data Trust and Security Mechanisms

Responsible Partner: CERTH
Contributing Partners: CERTH

Nature: ■ Report ☐ Prototype ☐ Demonstrator ☐ Other

Dissemination Level:

☐	PU :	Public
☐	PP :	Restricted to other programme participants (including the Commission Services)
■	RE :	Restricted to a group specified by the consortium (including the Commission Services)
☐	CO :	Confidential, only for members of the consortium (including the Commission Services)

Keyword List: web site, fact sheet, user guide, social media, project management

The eCOMPASS project (www.ecompass-project.eu) is funded by the European Commission, Information Society and Media Directorate General, Unit G4-ICT for Transport, under the FP7 Programme.

The eCOMPASS Consortium

Computer Technology Institute & Press "Diophantus" (CTI) (coordinator), Greece

Centre for Research and Technology Hellas (CERTH), Greece

Eidgenössische Technische Hochschule Zürich (ETHZ), Switzerland

Karlsruher Institut fuer Technologie (KIT), Germany

TOMTOM INTERNATIONAL BV (TOMTOM), Netherlands

PTV PLANUNG TRANSPORT VERKEHR AG. (PTV), Germany

Document history			
Version	Date	Status	Modifications made by
1.0	28.06.2013	First draft	Dionisis Kehagias
…			
…			
…			

Report Manager

- CERTH

List of Contributors

- CERTH
- CTI

Data Trust and Security Mechanisms

Objective

Data security mechanism was developed in order to ensure that information exchange between the modules of the CGM. The mechanism protect the transfer and the exchange of the data that are delivered both between the services and the end user. The mechanism allows only authenticated content providers to register services and devices using the CGM tool. Security Mechanism is about protecting data, that is, how to prevent unauthorized access or damage to data that is in storage or in transit.

In security terms, there are two important concepts: authentication and authorization. Understanding and the ways of implementing of these concepts can achieve the security effect of a project. Authentication is the way in which an entity (a user, an application, or a component) determines that another entity is who it claims to be. An entity uses security credentials to authenticate itself. The credentials might be a username and password, a digital certificate, or something else. When authentication is bidirectional, it is called mutual authentication. Authorization, also known as access control, is the means by which users are granted permission to access data or perform operations. After a user is authenticated, the user's level of authorization determines what operations the owner can perform.

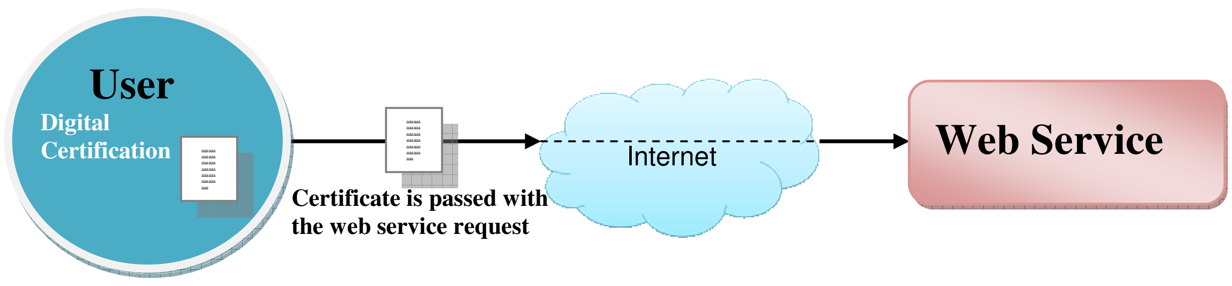
User's/ Services Trust and Security Techniques

Authentication techniques are found in many types of software ranging from databases to operating systems. The concept is that for each user of a system or service is assigned a unique username and the access to resources associated with that username is protected of a specific password. This method is ordinary for Web services because most users are already comfortable with and is easily understood by the users. The downside is that the credentials (username/password) usually be stored or transferred as clear text and the technique is unsecure. [1]

A more secure variation to this technique is to require a user identification code, called as Globally Unique IDentifier (GUID), and the resource that require authentication accepts the UserID in addition to its standard credentials. This approach is very hard to duplicate because of the GUIDs. GUIDs are usually stored as 128-bit values, and are commonly displayed as 32 hexadecimal digits with groups separated by hyphens, such as {54CE1221-5CFD-4321-B8BF-03007F21216C}.

Another option for securing web services is digital certificates. Digital certificates are small pieces of software installed on client machines that verify the client's identity (Figure A). This verification is done through a third-party that creates a unique certificate for every client machine using encryption. The certificate is then passed along when the client requests a web service. The web service checks for the presence of the digital certificate and reacts accordingly.

Figure A



Certificates, also called digital certificates, are electronic files that uniquely identify people and resources on the Internet. Digital certificates are typically used in an global prospect for the whole site, which means that it works for all content/services or for nothing. Each web service usually calls a single function, which verifies that a certificate was passed along with the request. If the function indicates that no certificate was passed, the web service fails and returns an appropriate error message. If the certificate is present, then functionality is executed normally. Digital certificates maintenance is easy and flexible. [2]

The security check for digital certificates doesn't occur until a Web method call is actually made. Thus, visitors can still view (the WSDL) pages for web services or associated web pages. However, the developer has to include code in each and every method that requires it, in order to avoiding that this method will be freely available to anyone who wants to use it.

Since the certificate is secure and unique, there's no need for the user to supply other kind of credentials such as username and password. The user/service can be authenticated with no additional effort on their part, all without sacrificing detailed auditing or method-level authorization. This is an important point toward user/ service friendliness. Also, certificates never obtain input from the user/service for authentication purposes and is only authenticating the machine/service, not the person using the machine.

The drawback of users' digital certificates is the restriction of the user to a specific machine. Even if someone installs a certificate on his own device (i.e. pc, mobile, etc), he has to install a second one in order to access the web service from another device. Although the fact that certificates never obtain input from the user/service for authentication purposes is an advantage, it can easily evolve into a disadvantage. If someone else try to use the device, the web service will assume that he is the original user who uses the service.

Secure Sockets Layer (SSL) is the most popular standard for securing Internet communications and transactions. Secure web applications use HTTPS (HTTP over SSL). The HTTPS protocol uses certificates to ensure confidential and secure communications between server and clients. In an SSL connection, both the client and the server encrypt data before sending it. Data is decrypted upon receipt. When a Web browser (client) wants to connect to a secure site, an SSL handshake happens, and the browser sends a message over the network requesting a secure session (typically, by requesting a URL that begins with https instead of http). The server responds by sending its certificate (including its public key). The browser verifies that the server's certificate is valid and is signed by a CA whose certificate is in the browser's database (and who is trusted). It also verifies that the CA certificate has not expired. If the certificate is valid, the browser generates a one time, unique session key and encrypts it with the server's public key. The browser then sends the encrypted session key to the server so that they both have a copy. The server decrypts the message using its private key and recovers the session key. The client has verified the identity of the Web site, and only the client and the Web server have a copy of the session key. From this point forward, the client and the server use the session key to encrypt all their communications with each other. Thus, their communications are ensured to be secure. The newest version of the SSL standard is called Transport Layer Security (TLS). [3]

Server's and CGM's Trust Abilities

Authentication

The Web Server of CGM is the GlassFish Server that is an open source software that provide important security issues and instructions for configuring and administering a server. The GlassFish Server is built on the Java security model, which uses a sandbox where applications can run safely, without potential risk to systems or users and system security affects all the applications in the GlassFish Server environment.

When an entity tries to access a protected resource, GlassFish Server uses the authentication mechanism configured for that resource to determine whether to grant access. For example, a user can enter a user name and password in a web browser, and if the application verifies those credentials, the user is authenticated. The user is associated with this authenticated security identity for the remainder of the session.

The Server supports several types of authentication, such as Basic, Form, Client-cert, Digest, JSR 196 Server Authentication Modules, Passwords, Single Sign-on, etc. Operations of these types [3]:

- **Basic**, which uses the server's built-in login dialog box via HTTP, but there is no user's credentials encryption unless using SSL. This type is not considered to be a secure method of user authentication unless used in conjunction with some external secure system such as SSL.
- **Form**, the application provides its own custom login and error pages via HTTP, and also there is no user's credentials encryption unless using SSL.
- **Client-cert**, where the server authenticates the client using a public key certificate via HTTPS (HTTP over SSL) with user's credentials encryption using SSL.
- **Digest**, that the server authenticates a user based on a username and a password and unlike to *basic* authentication, the password is never sent over the network and the use of SSL is optional.
- **JSR 196 Server Authentication Modules**, that defines a standard service-provider interface (SPI) for integrating authentication mechanism implementations in message processing runtimes. JSR 196 extends the concepts of the Java Authentication and Authorization Service (JAAS) to enable pluggability of message authentication modules in message processing runtimes. The standard defines profiles that establish contracts for the use of the SPI in specific contexts.
- **Passwords**, are the forefront of defense against unauthorized access to the components and data of GlassFish Server. GlassFish Server affords *Master Password* and *Keystores*. The master password is not tied to a user account and it is not used for authentication. Instead, GlassFish Server uses the master password only to encrypt the keystore and truststore for the DAS (Domain Administration Server) and instances. The DAS is a specially designated GlassFish Server instance that hosts administrative applications and is similar to any other GlassFish Server instance, except that the DAS has additional administration capabilities.
  The master password is used encrypt the keystore and truststore for the DAS and instances. The DAS needs the master password to open these stores at startup. GlassFish Server keeps the keystore and truststore for the DAS and instances in sync, which guarantees that all copies of the stores are encrypted with the same master password.
  However, GlassFish Server does not synchronize the master password itself, and it is possible thatthe DAS and instances might attempt to use different master passwords. Files that contain encoded passwords need to be protected using file system permissions.
- **Single Sign-on**, with which a user who logs in to one application becomes implicitly logged in to other applications that require the same authentication information. Single sign-on is based on groups. Single sign-on applies to web applications configured for the same realm and virtual server. The realm is defined by the realm-name element in the web.xml file.
  On GlassFish Server, single sign-on behavior can be inherited from the HTTP Service, enabled, or disabled. By default, it is inherited from the HTTP Service. If enabled, single sign-on is enabled for web applications on this virtual server that are configured for the same realm. If disabled, single sign-on is disabled for this virtual server, and users must authenticate separately to every application on the virtual server.

Authorization

User's authorization is based on roles. Roles, on GlassFish Server, a user's authorization is based on the user's role. A role defines which applications and what parts of each application users can access and what those users or groups can do with the applications. For example, in a personnel application of a transport company, all truckers might be able to see phone numbers and email addresses, but only managers have access to salary information. This application would define at least two roles; employee and manager. A role is different from a group in that a role defines a function in an application, while a group is a set of users who are related in some way. For the previous example of the company, the personnel application specify groups such as national and international truck drivers. Users in these groups are all employees (role).

Certificates and SSL

Certificates enable secure, confidential communication between two entities. There are different kinds of certificates:

- Personal certificates are used by individuals.
- Server certificates are used to establish secure sessions between the server and clients through secure sockets layer (SSL) technology.

Certificates are based on public key cryptography, which uses pairs of digital keys (very long numbers) to encrypt, or encode, information so the information can be read only by its intended recipient. The recipient then decrypts (decodes) the information to read it. A key pair contains a public key and a private key. The owner distributes the public key and makes it available to anyone. But the owner never distributes the private key, which is always kept secret. Because the keys are mathematically related, data encrypted with one key can only be decrypted with the other key in the pair.

Certificates are issued by a trusted third party called a Certification Authority (CA), and it validates the certificate holder's identity and signs the certificate so that it cannot be forged or tampered with. After a CA has signed a certificate, the holder can present it as proof of identity and to establish encrypted, confidential communications. Most importantly, a certificate binds the owner's public key to the owner's identity.

In addition to the public key, a certificate typically includes information such as the following:

- The name of the holder and other identification, such as the URL of the web server using the certificate, or an individual's email address
- The name of the CA that issued the certificate
- An expiration date

Certificates are governed by the technical specifications of the X.509 format. To verify the identity of a user in the certificate realm, the authentication service verifies an X.509 certificate, using the common name field of the X.509 certificate as the principal name.

A *certificate chain* is a series of certificates issued by successive CA certificates, eventually ending in a root CA certificate. Web browsers are preconfigured with a set of root CA certificates that the browser automatically trusts. Any certificates from elsewhere must come with a certificate chain to verify their validity.

When a certificate is first generated, it is a self-signed certificate. A self-signed certificate is one for which the issuer (signer) is the same as the subject (the entity whose public key is being authenticated by the certificate). When the owner sends a certificate signing request (CSR) to a CA, then imports the response, the self-signed certificate is replaced by a chain of certificates. At the bottom of the chain is the certificate (reply) issued by the CA authenticating the subject's public key. The next certificate in the chain is one that authenticates the CA's public key. Usually, this is a self-signed certificate (that is, a certificate from the CA authenticating its own public key) and the last certificate in the chain.

In other cases, the CA can return a chain of certificates. In this situation, the bottom certificate in the chain is the same (a certificate signed by the CA, authenticating the public key of the key entry), but the second certificate in the chain is a certificate signed by a different CA, authenticating the public key of the CA to which you sent the CSR. Then, the next certificate in the chain is a certificate authenticating the second CA's key, and so on, until a self-signed root certificate is reached. Each certificate in the chain (after the first) thus authenticates the public key of the signer of the previous certificate in the chain.

The GlassFish Server supports the SSL 3.0 and the TLS 1.0 encryption protocols. To use SSL, GlassFish Server must have a certificate for each external interface or IP address that accepts secure connections. The HTTPS service of most web servers will not run unless a certificate has been installed. For instructions on applying SSL to HTTP listeners. SSL and TLS support a variety of ciphers- cryptographic algorithm used for encryption or decryption- used to authenticate the server and client to each other, transmit certificates, and establish session keys.

Security Scheme and Roadmap of CGM

Certificate Generation and Processes

If private secure communication is that the authentication is required, can use only a self-signed certificate. By default, the keytool utility creates a keystore file in the directory where the utility is run. Before the keytool utility is running, shell environment must be configured so that the Java/bin directory is in the path, otherwise the full path to the utility must be present on the command line. In order to create a certification (not signed by a certificate authority) accomplish this the administrator must follow next steps from command line [3], [4]:

1. Change the path to the directory that contains the keystore and truststore files. Always generate the certificate in the directory containing the keystore and truststore files. The default is domain-dir/config (i.e. glassfish/domains/domain1/config).
2. Generate the certificate in the keystore file (keystore.jks), using the following command format:

   ```
   jdk/bin> keytool -genkey -alias <user_or_service_alias> -keyalg RSA -
   keypass changeit -storepass changeit -keystore keystore.jks
   ```

   Running this command, the administrator has to fill in the following screen arguments:

   ```
   What is your first and last name?

     [Unknown]: <i.e. CGM>

   What is the name of your organizational unit?

     [Unknown]: <i.e. eCompassDev>

   What is the name of your organization?

     [Unknown]: <i.e. eCompass>

   What is the name of your City or Locality?

     [Unknown]: <i.e. Thessaloniki>

   What is the name of your State or Province?

     [Unknown]: <i.e. Thessalonikis>

   What is the two-letter country code for this unit?

     [Unknown]: <i.e. GR>
   ```

   The keytool is asking to verify the correction of the answers:

   ```
   Is CN=CGM, OU=eCompassDev, O=eCompass, L=Thessaloniki,
   ST=Thessalonikis, C=GR correct?

     [no]:  yes
   ```

   Use any unique name as a <user_or_service_alias>. If the administrator has changed the keystore or private key password from the default (changeit), substitute the new password for changeit.
3. Export the generated certificate to the server.cer file (or client.cer if you prefer), using the following command format:

   ```
   jdk/bin> keytool -export -alias <user_or_service_alias> -storepass
   changeit -file server.cer -keystore keystore.jks
   ```

4. Create the cacerts.jks truststore file and add the certificate to the truststore, using the following command format:

   ```
   jdk1/bin> keytool -import -v -trustcacerts -alias
   <user_or_service_alias> -file server.cer -keystore cacerts.jks -
   keypass changeit
   ```

   If the keystore or private key password have been changed from the default (changeit), substitute the new password. Information about the certificate is displayed and a prompt appears asking if you
   want to trust the certificate. Information similar to the following is displayed:

   ```
   Enter keystore password:

   Re-enter new password:

   Owner: CN=CGM, OU=eCompassDev, O=eCompass, L=Thessaloniki,
   ST=Thessalonikis, C=GR
   ```

```
        Issuer: CN=CGM, OU=eCompassDev, O=eCompass, L=Thessaloniki,
        ST=Thessalonikis, C=GR

        Serial number: 3e4d3eb5

        Valid from: Fri Jun 21 13:15:27 EEST 2013 until: Thu Sep 19 13:15:27
        EEST 2013

        Certificate fingerprints:

                 MD5:  16:9A:B7:C6:77:8F:C4:3C:34:2A:83:C5:57:F8:F7:60

                 SHA1:
        25:24:C8:18:70:FF:91:04:DA:5C:AB:13:BB:EF:58:B7:25:48:3E:85

                 SHA256:
        4F:B5:BD:8D:F8:81:6B:50:A7:CD:9C:BC:A3:0E:83:0E:E5:CE:77:8D:15:

        09:9D:CB:FC:92:5D:D6:CB:EB:E4:7F

                 Signature algorithm name: SHA256withRSA

                 Version: 3


        Extensions:


        #1: ObjectId: 2.5.29.14 Criticality=false

        SubjectKeyIdentifier [

        KeyIdentifier [

        0000: A7 4B 3B 6D 5D 8C A6 F3   34 7C 65 27 68 AE E1
        FC  .K;m]...4.e'h...

        0010: 94 BC 57 FF                                        ..W.

        ]

        ]


        Trust this certificate? [no]:  yes

        Certificate was added to keystore

        [Storing cacerts.jks]
```

In order to display available certificates from a JKS Keystore, you can use the following command:

```
jdk/bin> keytool –list –v –keystore <keystore.file> –storepass
<keystore.pass>
```

and for displaying certificate information from a JKS Keystore the appropriate command is:

```
jdk/bin> keytool –list –v –alias ${cert.alias} –keystore ${keystore.file} –
storepass ${keystore.pass}
```

Delete the default self-signed certificate as follow:

```
jdk/bin> keytool -delete -alias s1as -keystore keystore.jks -storepass
<store_passwd>
```

where <store_passwd> is the password for the keystore (i.e. mypass) and s1as is the default alias of the GlassFish Server keystore.

Generation of a new key pair for the application server is like:

```
jdk/bin> keytool -genkeypair -keyalg <key_alg> -keystore keystore.jks -
validity <val_days> -alias s1as
```

where <key_alg> is the algorithm to be used for generating the key pair (i.e. RSA, base64Binary, etc.), and <val_days> is the number of days that the certificate should be considered valid. In addition to generating a key pair, the command wraps the public key into a self-signed certificate and stores the certificate and the private key in a new keystore entry identified by the alias.

For HTTPS hostname verification, it is important to ensure that the name of the certificate (CN) matches the fully-qualified hostname of your site (fully-qualified domain name). If the names do not match, clients connecting to the server will see a security alert stating that the name of the certificate does not match the name of the site.

Another variation of execution of the command keytool follows, and this variation create a digital signature:

```
jdk/bin> keytool –certreq –keyalg RSA –alias <user_or_service_alias> –file
certreq.csr –keystore keystore
```

Running this command, the administrator has to fill in the following screen arguments:

```
Enter keystore password:

Enter key password for <user_or_service_alias>
```

Information similar to the following is displayed using the command type:

```
jdk/bin> type certreq.csr

-----BEGIN NEW CERTIFICATE REQUEST-----
```

```
MIIC2TCCAcECAQAwZDELMAkGA1UEBhMCR1IxDzANBgNVBAgTBkFjaGFpYTEQMA4GA1UEBxMHVXBh
dHJhczENMAsGA1UEChMEQ0VJRDERMA8GA1UECxMIZUNvbXBhc3MxEDAOBgNVBAMTB0NHTVRlc3Qw
ggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQCv4cLCZooEgIHWQGLD48lz/Y5FX2lG/bdQ
ZvwdK8oh4kAPL9bZ3ApQb23PMcMjzp1PKwwvGd51d2A1B03h/thn70JTMUvcF3oE1232Cv7pEh1R
+5jY8AGFizJe/C00hndKrqP0cpxBx/p8O0QoTh0mLGYtFeBHv/BKKlDMpQDIzeN9LVXByJkcW4vv
EV7A6eM5NwHGQd3LlW9dynsHV72yS7TnGBXV1DVI57X/PRoPg2Fh2MAX5jkj2GZnF5/wcrTyuHo6
tv6gAMq+1F9mXB2T6IxMMMwhXmUBcVHWuxPaV/f8SA95byInTgreAtNuhPLF/p9/sPJlE6VmNqk+
qIILAgMBAAGgMDAuBgkqhkiG9w0BCQ4xITAfMB0GA1UdDgQWBBS2014A2jSpKtTQbEeg/RlAzw9C
GzANBgkqhkiG9w0BAQsFAAOCAQEAGncARSZbGL6IV4DwoTsXVl6FCKUXxqrKQD385z0cpvRshBLi
Rc8fTe80EpXgM7YmJJW2p/5ti46GyWNV5Hi2Jwf4XYZxqMcR2iXFcPgb2m0gamf7FRt8FLvCQ+fQ
BkmoN8/XdqyvWTcnOPdrp8G9E4ISpAMRl0HMP90PwD2hs/sckZzsjoWa+QeHfIQzFT9ASIJ5aYtZ
52S1pt0YKZpmEPDQCOwuEKMhkgBY+r4Ux/ytv0Djq4BHpXHPXYuZFiD9DG3fUv7+KhYMFJODf6aO
l0LroFc4+byQEWijL/jrB2cbuAT9WiRts/dMdyREojcTvknqAB3hVQ3dWtVvv03xdA==
```

```
-----END NEW CERTIFICATE REQUEST-----
```

To apply any changes, you have to restart GlassFish Server. At this point a certificate that can be used for digital signatures has been obtained.

Services Certificate Processing

The next step is to sign a document. There are a number of ways to do this and the process will vary based on how your original document is set up. Signing basically involves picking one or more sections in the document that are marked for signing, getting them signed and then embedding the signature into the documents [5].

Web Services Policy (WS-Policy) is a standards-based framework for defining a Web service's security constraints and requirements. It expresses security constraints and requirements in a collection of XML statements called policies, each of which contains one or more assertions. WS-Policy assertions are used to specify a Web service's requirements for digital signatures and encryption, along with the security algorithms and authentication mechanisms that it requires, however the WS-Policy specification has not been fully standardized. WS-Policy policies can be included directly in a WSDL document or included by reference, and a WSDL document may import other WSDL documents that contain or refer to WS-Policy policies. An XML file that contains these policies can be used by multiple proxy services or business services [6], [7].

WebLogic Web Services are implemented according to the Enterprise Web Services 1.1 specification (JSR-921), which defines the standard J2EE runtime architecture for implementing Web Services in Java. The specification also describes a standard J2EE Web Service packaging format, deployment model, and runtime services, all of which are implemented by WebLogic Web Services.

WebLogic provides message-level security for web services through an implementation of the WS-Security web service security standard. WS-Security gives the opportunity of securing the SOAP messages passed between web services using (1) security tokens, (2) digital signatures, and (3) encryption. Although supports both transport and message-level security, it is generally not necessary to use both sorts of security to secure a web service. In most cases, developers should choose one or the other type of security to secure their web services.

Security tokens are credentials used for authentication, authorization, or both. The implementation supports two types of tokens. (1) Username and password tokens, and (2) X509 Binary Security Tokens. Digital signatures are used for two purposes: (1) to authenticate the identity of the sender and (2) to ensure the integrity of SOAP messages. If any part of an incoming SOAP message has been changed in transport, the signature validation performed by the recipient will fail. In WebLogic, if XML signatures is required for incoming SOAP messages, the SOAP body must be digitally signed to be processed by the web service. By default, digital signatures are applied only to the body of outgoing SOAP messages. The developer must specifically provide the signing of elements in the header (<addtionalSignedElements>). Encryption is used to encrypt either the body of the SOAP message, the header, or both. If a web service requires encryption for incoming messages, then, at a minimum, the body of incoming SOAP messages must be encrypted. For outgoing SOAP messages, encryption is applied only to the SOAP body by default and must specifically provide for the encryption of elements in the header (<addtionalEncryptedElements>). Keys used in WebLogic's implementation of WS-Security must be RSA keys [8].

Web service security is controlled through WSSE policy files. WSSE policy files are XML files with a .WSSE file extension. To secure a web service with web service security, it is necessary to create a WSSE policy file and that file ought to be associated with the web service. All outbound and inbound SOAP messages are processed according to the policy called for in the WSSE file. Inbound messages are first checked for the necessary security measures called for in the policy file. If the inbound message is found to be appropriately secured, then the SOAP message, cleaned of its security enhancements, is passed to the web service for normal processing. Outbound messages go through the reverse process: they are enhanced with the security measures called for in the policy file before they sent out over the wire. To access a web service secured with WS-Security, create a policy file and associate that file with the web service control. The policy file that is associated with a web service's control should match the policy file of the target web service. If the target web service requires encrypted incoming messages, then a control file targeting that web service should encrypt messages before they are sent to the web service [8].

The WebLogic Web Services runtime environment recognizes two types of WS-Policy statements:

- **Concrete** WS-Policy statements specify the security tokens that are used for authentication, encryption, and digital signatures and can create concrete WS-Policy statements if it is known at design time the type of authentication that you want to require; whether multiple private key and certificate pairs from the keystore are going to be used for encryption and digital signatures; and so on.
- **Abstract** WS-Policy statements that do not specify security tokens.

AquaLogic comprised a software suite developed by Oracle (BEA Systems) for managing service-oriented architecture (SOA). AquaLogic Service Bus is a product that has operational service-management. It allows the interaction between services, routing relationships, transformations, and policies. includes three XML files that contain simple, abstract WS-Policy policies:

- Auth.xml, that contains a policy that requires Web service clients to authenticate.
- Encrypt.xml, which contains a policy that requires clients to encrypt the SOAP body (i.e using algorithm RSA).
- Sign.xml, it contains a policy that requires clients to sign the SOAP body. It also requires that the WS-Security engine on the client add a signed timestamp to the wsse:Security header, which prevents certain replay attacks. All system headers are

also signed. The digital signature algorithm is RSA-SHA1. Exclusive XML canonicalization is used.

The system headers are:

- wsrm:SequenceAcknowledgement | AckRequested | Sequence
- wsa:Action | From | To | FaultTo | MessageID | RelatesTo | ReplyTo
- wsu:Timestamp
- wsax:SetCookie

and the namespace prefixes correspond to the:

- http://schemas.xmlsoap.org/ws/2005/02/rm
- http://schemas.xmlsoap.org/ws/2004/08/addressing
- http://schemas.xmlsoap.org/ws/2002/07/utility
- http://schemas.xmlsoap.org/ws/2004/01/addressingx

It is recommended to use these pre-packaged policies whenever possible. However, they cannot be used in some cases that if the provided WS-Policy statements don't meet the security needs. Instead, it is possible for the developer to write his own WS-Policy statements (custom WS-Policy statements), but cannot modify the AquaLogic Service Bus WS-Policy statements. The developer can either write custom WS-Policy statements directly in a Web service's WSDL document or, in case of reusing the statements in multiple Web services, the developer can write them in a separate XML file and refer to them from the WSDL documents.

To apply encryption policy for a service and if we want the service's messages to be encrypted, it is necessary to create a custom WS-Policy. The policy must be concrete (it must contain the encryption certificate instead of using a certificate from a proxy service provider) and it must be located directly in a WSDL document instead of being included by reference. It would require messages to a service to be encrypted if the proxy service that sends messages to the service is a pass-through proxy service. That is, the proxy service that receives messages from a client does not process the SOAP message. Instead, the proxy service routes the message to the service, and the service takes on the responsibility of Web Services Security. In the following listing is shown the encrypting of the Body with a Concrete Policy, Embedding the Policy in the WSDL Document:

```
<definitions name="WssServiceDefinitions"
targetNamespace="http://com.bea.alsb/tests/wss"
xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"...>

        <wsp:UsingPolicy xmlns:n1="http://schemas.xmlsoap.org/wsdl/"
n1:Required="true"/>


        <!-- The policy provides a unique ID -->

        <wsp:Policy wsu:Id="myEncrypt.xml">

                <wssp:Confidentiality

                        xmlns:wssp="http://www.bea.com/wls90/security/policy">

        <wssp:KeyWrappingAlgorithm URI="http://www.w3.org/2001/04/xmlenc#rsa-
1_5"/>

        <!-- Require the user name and password in the security header to be
encrypted -->

        <wssp:Target>

                <wssp:EncryptionAlgorithm
URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>

        <wssp:MessageParts
Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">

                wsp:Body()

        </wssp:MessageParts>
        </wssp:Target>


        <!-- Embed the token type and encryption certificate -->

        <wssp:KeyInfo>

                <wssp:SecurityToken TokenType="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"/>

                <wssp:SecurityTokenReference>

                <wssp:Embedded>

                        <wsse:BinarySecurityToken
EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-
message-security-1.0#Base64Binary" ValueType="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd">

                                MIICfjCCAeegAwIBAgIQV/PDyj3...

                        </wsse:BinarySecurityToken>
                </wssp:Embedded>

                </wssp:SecurityTokenReference>

        </wssp:KeyInfo>

        </wssp:Confidentiality>

</wsp:Policy>

<binding name="WssServiceSoapBinding" type="tns:WssService">

        <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>

        <operation name="getPurchaseOrder">

        <soap:operation soapAction="" style="document"/>

        <input>

        <soap:body parts="parameters" use="literal"/>


        <!-- Use a URI fragment to refer to the unique policy ID -->

        <wsp:Policy>

                <wsp:PolicyReference URI="#myEncrypt.xml"/>

                </wsp:Policy>

        </input>

<output>

        <soap:body parts="parameters" use="literal"/>

</output>

</operation>

</binding>

...

</definitions>
```

This listing is a WSDL document that contains a concrete policy. As is shown:

- The policy requires clients to encrypt the message body,
- the KeyInfo element specifies the type of token that a client must provide to is the parent element that is used to describe and embed the encryption certificate. The BinarySecurityToken element contains the encryption certificate. If the certificate is in PEM format, the content of the PEM file (without the PEM prefix and suffix) is the

encoded representation of the certificate. If the encryption certificate is stored in a JDK keystore, it can easily be exported to a PEM file.
- The policy provides a unique ID and the WSDL uses a URI fragment to refer to the ID.

To attach WS-Policy statements to a WSDL document for a Web service, in case of creation a custom WS-Policy in a separate XML file (after adding the custom WS-Policy file as a resource in the AquaLogic Service Bus domain), add the following child element in the <definitions> element of the WSDL document:

```
<wsp:UsingPolicy wsdl:Required="true"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"/>
```

where the wsdl:required="true" attribute ensures that proxy services and services are capable of processing the policy attachments.

Within each element in the WSDL document that you want to secure, it is required to determine the URI of the WS-Policy statements that use and specify the URI in the WSDL document. To determining the URI of a WS-Policy Statement for the AquaLogic Service Bus, the URIs are always as follows:

- policy:Auth.xml
- policy:Encrypt.xml
- policy:Sign.xml

For WS-Policy statements that are located directly in the WSDL document, the URI is as follows:

```
#policy-ID where policy-ID is the value of the policy's wsu:ID attribute
```

For WS-Policy statements that you created in a separate XML file and added as resources to AquaLogic Service Bus, the URI is as follows:

```
policy:policy-ID
```

where policy-ID is the value of the policy's wsu:ID attribute (which the developer specified in the policy's XML file).

Using one of the following techniques to specify the URI of a WS-Policy Statement in a WSDL document (described in http://www.w3.org/TR/wsdl):

- PolicyURIs attribute, if the WSDL schema allows attribute extensibility for the element that you want secure, add the PolicyURIs global attribute to the element.
- Nested <Policy> element, if the WSDL schema allows element extensibility for the element that you want to secure, add <Policy> as a global child element. For each WS-Policy that you want to use, add one <PolicyReference> element as a child of the <Policy> element.

References

[1]. Jaroslav Imrich, APACHE web server and SSL authentication (http://linuxconfig.org/apache-web-server-ssl-authentication)

[2]. IBM, Web services authentication method overview, i5/OS Information Center, Version 5 Release 4

[3]. GlassFish Server Open Source Edition, Security Guide, Release 4.0, May 2013

[4]. GlassFish Server Open Source Edition, Administration Guide, Release 4.0, May 2013

[5]. http://www.w3.org/TR/xmldsig-core/

[6]. XML Signature Syntax and Processing Version 1.1, W3C Recommendation. April 2013

[7]. Custom WS-Policies (http://docs.oracle.com/cd/E13171_01/alsb/docs26/consolehelp/policies.html)

[8]. Programming Web Services for WebLogic Server (http://docs.oracle.com/cd/E13222_01/wls/docs92/webserv/security.html)